



# Design and Implementation of rowe, a Web-Friendly Communication Library

Ludovic Courtès

**TECHNICAL  
REPORT**

**N° 452**

Janvier 2015

Project-Team Indes





## Design and Implementation of rowe, a Web-Friendly Communication Library

Ludovic Courtès

Project-Team Indes

Technical Report n° 452 — Janvier 2015 — 7 pages

**Abstract:** The INDES project-team of Inria has been developing HOP, a multi-tier language for Web programming. As part of the RAPP FP7 European project, the team has set out to use HOP as the *lingua franca* of the robotics applications developed within that project. Part of the challenge lies in the integration of existing robotics code, written using ROS or custom libraries, with HOP-based application.

This document reports on the implementation of rowe, a communication library designed to fill the gap between low-level robotics C components on one hand, and other C, C++, ROS, or HOP components on the other. The library aims to be a lightweight, high-performance, “Web-friendly” communication library. It implements a socket-like interface that allows programs to exchange JSON objects over WebSockets. We describe the rationale, design, and implementation of rowe.

**Key-words:** communication library, websocket, JSON

RESEARCH CENTRE  
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour  
33405 Talence Cedex

## Conception et mise en œuvre de rowe, une bibliothèque de communication orientée Web

**Résumé :** L'équipe-projet Inria INDES développe HOP, un langage à plusieurs niveaux pour la programmation Web. Dans le cadre du projet européen FP7 RAPP, l'équipe s'est donné pour objectif d'utiliser HOP comme *lingua franca* des applications robotiques développées dans le projet. Un des défis à relever est l'intégration de code robotique existant, utilisant ROS ou des bibliothèques dédiées, avec des applications en HOP.

Ce document décrit rowe, une bibliothèque de communication visant à combler le vide entre d'un côté des composants robotiques bas niveau écrits en C, et d'un autre côté des composants écrits en C++, ROS ou HOP. L'objectif est de fournir une bibliothèque de communication légère, haute performance et qui s'intègre facilement à l'environnement Web. Elle met en œuvre une interface de type *socket* permettant d'échanger des objets JSON sur des WebSockets. Nous décrivons les motivations, les choix de conception, et la mise en œuvre de rowe.

**Mots-clés :** bibliothèque de communication, websocket, JSON

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Rationale</b>	<b>3</b>
2.1	Design Goals . . . . .	4
2.2	Programming Interface . . . . .	4
<b>3</b>	<b>Implementation</b>	<b>6</b>
3.1	Service Thread . . . . .	6
3.2	Remote Procedure Calls . . . . .	6
3.3	Time-to-Live . . . . .	7
<b>4</b>	<b>Summary</b>	<b>7</b>

## 1 Introduction

RAPP<sup>1</sup> is a project of the European Commission’s Seventh Framework Program (FP7). It seeks to deliver a software platform including robotic applications for social inclusion. The goal is to deliver robotic applications (or “RApps”) for a variety of a robots—ranging from humanoid robots, to an instrumented walking aid for elderly people—that will often need to interact with Web services, for instance to access user profile information. Those applications typically use a variety of programming languages and technologies, such as C, C++, Python, ROS<sup>2</sup>, and JavaScript.

HOP is a Web programming framework [4] that makes it easy to use the protocols and formats of the Web: HTTP, WebSocket [1], JSON, etc. HOP also comes with a package management tool, called Hz, that simplifies the installation of HOP applications. HOP was chosen as the tool to build the infrastructure for RAPP’s application store and on-line services, as well as the tool to connect application components together [3].

To that end, we developed two pieces of software: a HOP interface to ROS, and rowe, a C library to connect with embedded robotics software that does not use ROS—such as the Assistive Navigation Guide (ANG) family of walking aids developed by Inria’s Héphaïstos team<sup>3</sup>.

The next section describes the goals and design choices that we set out for rowe. Section 3 gives an overview of its implementation. Section 4 concludes.

## 2 Rationale

The rowe library aims to connect C robotics software with components that communicate using the ROS protocols or Web protocols.

---

<sup>1</sup>See the RAPP Web site at <http://rapp-project.eu/>.

<sup>2</sup>ROS Web site at <http://www.ros.org/>.

<sup>3</sup>See an overview of the ANG family of walking aids at <https://pal.inria.fr/research/themes/rehabilitation-transfer-and-assistance-in-walking/walking-aids/>.

## 2.1 Design Goals

We set out a number of design goals for rowe:

1. Robotics software using rowe is going to send status updates to other RAPP components at a possibly high rate (for instance, the speed and location of the robot, similar to ROS “topics”), and it must receive and process requests from other components in a timely fashion (such requests may include an emergency stop, for instance, similar to ROS services.) Thus, rowe must guarantee low latency and high throughput.
2. Programs using rowe are meant to be connected primarily with HOP programs, so rowe must be a “native speaker” of the Web protocols and formats.
3. It must be possible using rowe to exchange typed and structured messages, such as strings, numbers, records, lists, and so on.
4. The application programming interface (API) of rowe should match the programming style and expectations of low-level robotics developers. Informally, that means that it should be as little disruptive as possible.

Being a native of the Web meant that rowe’s transport layer should be based on HTTP, which also has the advantage of being usually allowed traffic through firewalls. Of course using HTTP alone to exchange messages, for instance in a ReST fashion, would incur too much overhead: HTTP connections would regularly need to be instantiated, which unacceptably increases latency, and HTTP GET requests may incur too much bandwidth overhead.

For that reasons, we chose to use WebSockets as the transport later [1]. WebSockets is an HTTP extension that provides a reliable, bidirectional communication channel that can be used to transfer arbitrary payloads, similar to TCP.

JSON, for JavaScript Object Notation [2], came up as the obvious choice for the message format. It meets our requirements as a mechanism to encode structured and typed messages, it is the natural way to represent data in HOP programs, and has efficient parsers and serializers.

The last design goal is more subjective. In our view, matching the programming style of low-level robotics developers meant a few things. First, the API should be usable in single-threaded programs. Second, it should not expose a full-blown event loop framework as commonly found in object-oriented libraries such as GLib. Those frameworks are generally complex, and they impose *inversion of control* (IoC) through a heavy use of callbacks: that essentially forces developers to write in continuation-passing style (CPS), which is both verbose and difficult to work with. Instead, we want to allow a direct programming style. This has been the main choice driving the design of the programming interface.

## 2.2 Programming Interface

The bulk of rowe’s programming interface has purposefully been kept minimal and simple.

In rowe version 1, connections are modeled by an *endpoint*. A rowe program can only be connected to one peer at a time (see Section Conclusion for a

discussion and desired changes to this approach.) A rowe program can be an HTTP server:

```
struct rowe_endpoint *endpoint;
endpoint = rowe_open_local_endpoint (8080);
```

or it can be an HTTP client:

```
endpoint = rowe_open_remote_endpoint ("hop.inria.fr", 8080);
```

The API to send and receive messages is the same regardless of whether the program is a server or a client.

Messages are JSON objects, as implemented by the JSON-C library<sup>4</sup> sent to an endpoint using the `rowe_send` function. Callers can specify a time-to-live (TTL) for the message: if no peer was connected after the TTL has expired, the message is discarded and not sent. This is useful for periodic messages such as updates on the robot's status, akin to ROS topics: an old update is not valuable and can be discarded, allowing the peer to get fresher updates instead.

```
extern int rowe_send (const struct rowe_endpoint *endpoint,
                     const struct json_object *obj,
                     long ttl);
```

The `rowe_send` function is synchronous and blocks until the message has either been sent, or has been discarded. Alternately, the `rowe_async_send` function is non-blocking, any may typically be used when sending messages containing status updates.

Similarly, the `rowe_receive` functions blocks until a message is received or the user-specified timeout has expired, and returns a pointer to a `json_object` structure or NULL.

Programs may also perform remote procedure calls (RPCs), using the `rowe_invoke` function:

```
extern struct json_object *
rowe_invoke (const struct rowe_endpoint *endpoint,
             struct json_object *obj, long timeout);
```

The function sends the given JSON object, which denotes a procedure invocation, blocks until a reply has been received, and returns it, unless the given timeout has expired. The actual format of the JSON object representing the procedure call is at the user's discretion. An example JSON-formatted service invocation may look like this:

```
{
  "service": "add-two-numbers",
  "a": 38,
  "b": 4
}
```

---

<sup>4</sup>The JSON-C library: <https://github.com/json-c/json-c/wiki>.

Lastly, `rowe` programs can reply to RPCs, using the `rowe_reply` function or using `rowe_async_reply`, its non-blocking counterpart.

To facilitate the creation of JSON objects representing key/value associations, the `rowe_message` convenience function is provided. For instance, the message shown above may be instantiated with the following call:

```
struct json_object *invocation;

invocation = rowe_message ("service",
    json_type_string, "add-two-numbers",
    "a", json_type_int, 38,
    "b", json_type_int, 4, NULL);
```

## 3 Implementation

The implementation of the above API goes along the following lines.

### 3.1 Service Thread

`rowe` builds upon the JSON-C<sup>4</sup> and libwebsockets<sup>5</sup> libraries. Since it does not expose an event loop interface, the actual event loop runs in a dedicated service thread, which is spawned when the endpoint is opened. The service thread polls for connection requests and for “in” and “out” events on open connections.

The service thread adds incoming messages on a queue that is checked by functions such as `rowe_receive`. When `rowe_send` and similar functions are called from the user thread, they add the given message to an outgoing message queue, which the service thread checks when the connection is ready to accept outgoing messages. When the service thread accesses the outgoing message queue, it deletes any messages whose TTL has expired.

The `rowe_async_send` function is the simplest: it just adds a message to the outgoing message queue and returns immediately. Conversely, the `rowe_send` and `rowe_receive` functions need to synchronize with the service thread. `rowe_receive` checks for message in the incoming message queue; when the message queue is empty, it waits on a condition variable associated with it. `rowe_send` works by passing a *notification* object, which essentially bundles together a condition variable and a return value, which the service thread notifies when the message is discarded due to TTL expiration, or once it has been sent.

### 3.2 Remote Procedure Calls

RPC replies need special treatment: when `rowe_invoke` is used, unrelated messages may be received after the invocation message has been sent and before the reply has been received; yet, `rowe_invoke` must return the RPC reply, not another message that happened to be received first.

To address that, `rowe` takes several steps. First, it requires invocation messages to be JSON objects (key/value associations) and, upon invocation, it automatically adds them a `message_id` entry whose value is a unique identifier, allowing the invocation to be distinguished from other invocations of the same

---

<sup>5</sup>The libwebsockets library: <http://libwebsockets.org/>.



remote procedure. RPC replies must also be JSON objects, and they must have a `in_reply_to` entry whose value is the `message_id` of a previous invocation message.

Second, `rowe` maintains a table that allows it to match RPCs with replies, and to wake up the user thread that is waiting in `rowe_invoke`. The table is essentially a list of pairs of `message_id` values and corresponding notification object that allows the user thread, which may be waiting in `rowe_invoke`, to be woken up.

### 3.3 Time-to-Live

Since the service thread must periodically removed expired messages both from the incoming and the outgoing message queues, this operation must be efficient. To that end, the message queue structure is (1) a doubly-linked list, which provides for constant-time deletion, and (2) it can be viewed both as a queue (FIFO) and as a list of messages ordered by expiration date, which makes it more efficient.

## 4 Summary

The `rowe` library provides a simple programming interface for connected components. It is well suited for the RAPP project where it allows low-level robotics software to communicate with HOP programs using Web protocols, and with good performance, notably on low-end embedded ARM-based devices.

As of this writing, version 2 of `rowe` is being developed. The main goal is to allow users to distinguish between an endpoint and an established connection, and to support connections with multiple peers.

The `rowe` library is free software, available from <https://github.com/rapp-project/rowe> and from <ftp://ftp-sop.inria.fr/index/rapp/rowe/>.

## References

- [1] I. Fette and A. Melnikov. RFC 6455—the WebSocket protocol, December 2011.
- [2] ECMA International. Standard ECMA-404—the JSON data interchange format, October 2013.
- [3] Fotis Psomopoulos, Emmanouil Tsardoulis, Alexandros Giokas, Cezary Zielinski, Vincent Prunet, Ilias Trochidis, David Daney, Manuel Serrano, Ludovic Courtès, Stratos Arampatzis, and Pericles A. Mitkas. RAPP system architecture. In *Workshop on Assistance and Service Robotics in a Human Environment in IROS*, September 2014.
- [4] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, USA, October 2006.



**RESEARCH CENTRE  
BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour  
33405 Talence Cedex

Publisher  
Inria  
Domaine de Voluceau - Rocquencourt  
BP 105 - 78153 Le Chesnay Cedex  
[inria.fr](http://inria.fr)

ISSN 0249-0803